PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 17/50

A1

(11) International Publication Number: WO 00/38087

(43) International Publication Date: 29 June 2000 (29.06.00)

(21) International Application Number: PCT/GB99/04338

(22) International Filing Date: 21 December 1999 (21.12.99)

(30) Priority Data:
9828381.5 22 December 1998 (22.12.98) GB

(71) Applicant *(for all designated States except US)*: DASH TECHNOLOGIES LIMITED [GB/GB]; 52 New Inn Hall Street, Oxford OX1 2QA (GB).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: SAUL, Jonathan, Martin [GB/GB]; 205 Poplar Grove, Kennington, Oxford OX1 5QT (GB). AUBURY, Matthew, Philip [GB/GB]; 148 Godstow Road, Wolvercote, Oxford OX2 8PG (GB).

(74) Agents: NICHOLLS, Michael, John et al.; J.A. Kemp & Co., 14 South Square, Gray's Inn, London WC1R 5LX (GB).

(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published

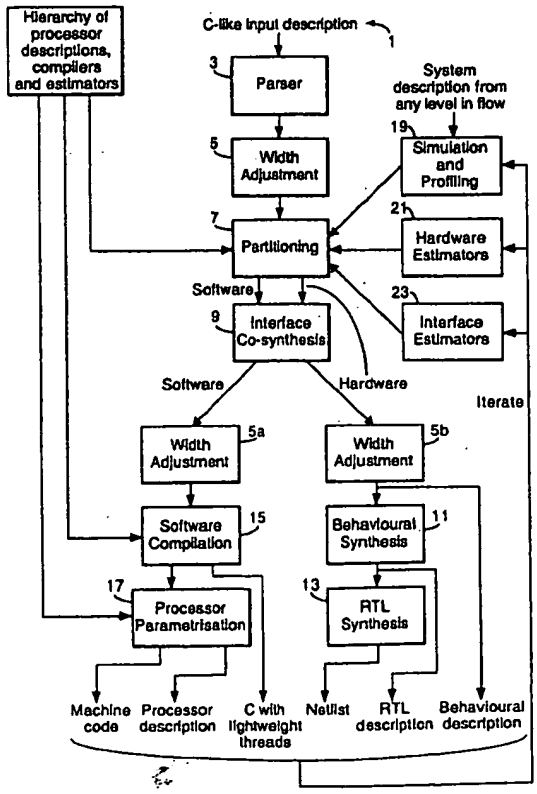
With international search report.

Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: HARDWARE/SOFTWARE CODESIGN SYSTEM

(57) Abstract

A hardware/software codesign system for making an electronic circuit which includes both dedicated hardware and software controlled resources. The codesign system receives a behavioural description of the target electronic system and automatically partitions the required functionality between hardware and software, while being able to vary the parameters (e.g. size or power) of the hardware and/or software. Thus, for instance, the hardware and the processor for the software can be formed on an FPGA, each being no bigger than is necessary to perform the desired functions. The codesign system outputs a description of the required processor (which can be in the form of a net list for placement on the FPGA), machine code to run on the processor, and a net list or register transfer level description of the necessary hardware. It is possible for the user to write some parts of the description of the target system at register transfer level to give closer control over the operation of the target system, and the user can specify theprocessor or processors to be used, and can change, for instance, the partitioner, compilers or speed estimators used in the codesign system. The automatic partitioning may be performed by using a genetic algorithm which estimates the performance of randomly generated different partitions and selects an optimal one of them.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

HARDWARE/SOFTWARE CODESIGN SYSTEM

The present invention relates to a system for designing and producing an electronic circuit having a desired functionality and comprising both hardware which is dedicated to execution of certain of the functionality and a software-controlled machine for executing the remainder of the functionality under the control of suitable software.

It is well known that software-controlled machines provide great flexibility in that they can be adapted to many different desired purposes by the use of suitable software. As well as being used in the familiar general purpose computers, software-controlled processors are now used in many products such as cars, telephones and other domestic products, where they are known as embedded systems.

However, for a given function, a software-controlled processor is usually slower than hardware dedicated to that function. A way of overcoming this problem is to use a special software-controlled processor such as a RISC processor which can be made to function more quickly for limited purposes by having its parameters (for instance size, instruction set etc.) tailored to the desired functionality.

Where hardware is used, though, although it increases the speed of operation, it lacks flexibility and, for instance, although it may be suitable for the task for which it was designed it may not be suitable for a modified version of that task which is desired later. It is now possible to form the hardware on reconfigurable logic circuits, such as Field Programmable Gate Arrays (FPGA's) which are logic circuits which can be repeatedly reconfigured in different ways. Thus they provide the speed advantages of dedicated hardware, with some degree of flexibility for later updating or multiple functionality.

In general, though, it can be seen that designers face a problem in finding the right balance between speed and generality. They can build versatile chips which will be software controlled and thus perform many different functions

relatively slowly, or they can devise application-specific chips that do only a limited set of tasks but do them much more quickly.

A compromise solution to these problems can be found in systems which combine both dedicated hardware and also software. The hardware is dedicated to particular functions, e.g. those requiring speed, and the software can perform the remaining functions. The design of such systems is known as hardware-software codesign. Within the design process, the designer must decide, for a target system with a desired functionality, which functions are to be performed in hardware and which in software. This is known as partitioning the design. Although such systems can be highly effective, the designer must be familiar with both software and hardware design. It would be advantageous if such systems could be designed by people who have familiarity only with software and which could utilise the flexibility of configurable logic resources.

The present invention provides a hardware/software codesign system which can target a system in which the hardware or the processors to run the software can be customised according to the functions partitioned to it. Thus rather than the processor or hardware being fixed (which effectively decides the partitioning), the codesign system of this invention includes a partitioning means which flexibly decides the partitioning while varying the parameters of the hardware or processor to obtain an efficient overall design that is close to optimal.

In more detail it provides a codesign system for producing a target system having resources to provide specified functionality by:

(a) operation of dedicated hardware; and

(b) complementary execution of software on software-controlled machines;

the codesign system comprising means for receiving a specification of said functionality; partitioning means for partitioning implementation of said functionality between (a) and (b) and for customising said hardware and /or said machine in accordance with the selected partitioning of the functionality.

Thus the target system is a hybrid hardware/software system. It can be formed using configurable logic resources in which case either the hardware or the

processor, or both, can be formed on the configurable logic resources (e.g. an FPGA).

In one embodiment of the invention the partitioning means uses a genetic algorithm to optimise the partitioning and the parameters of the hardware and the processor. Thus, it generates a plurality of different partitions of the functionality of the target system (varying the size of the hardware and/or the processor between the different partitions) and estimates the speed and size of the resulting system. It then selects the best found partitioning on the basis of the estimates. In the use of a genetic algorithm, a variety of partitions are randomly generated, the poor ones are rejected, and the remaining ones are modified by combining aspects of them with each other to produce different partitions. The speed and size of these are then assessed and the process can be repeated until a sufficiently good partition is produced.

The invention is applicable to target systems which use either customizable hardware and a customizable processor, or a fixed processor and customizable hardware, or fixed hardware and a customizable processor. Thus the customizable part could be formed on an FPGA, or, for instance, an ASIC.

The system may include estimators for estimating the speed and size of the hardware and the software controlled machine and may also include an interface generator for generating interfaces between the hardware and software. In that case the system may also include an estimator for estimating the size of the interface. The partitioning means calls the estimators when deciding on the quality of each possible partitioning.

The software-controlled machine can comprise a CPU and the codesign system comprises means for generating a compiler for the CPU as well as means for describing the CPU where it is to be formed on customizable logic circuits.

The codesign system can further comprise a hardware compiler for producing from those parts of the specification partitioned to hardware a register transfer level description for configuring configurable logic resources (such as an FPGA). It can further include a synthesizer for converting the register transfer level description into a net list.

The system can include a width adjuster for setting a desired data word size, and this can be done at several points in the desired process as necessary.

Another aspect of the invention provides a hardware/software codesign system which receives a specification of a target system in the form of a behavioural description, i.e. a description in a programming language such as can be written by a computer programmer, and partitions it and compiles it to produce hardware and software.

The partitioning means can include a parser for parsing the input behavioural description. The description can be in a familiar computer language such as C, supplemented by a plurality of predefined attributes to describe, for instance, parallel execution of processes, an obligatory partition to software or an obligatory partition to hardware. The system is preferably adapted to receive a declaration of the properties of at least one of the hardware and the software-controlled machine, preferably in an object-oriented paradigm. It can also be adapted such that some parts of the description can be at the register transfer level, to allow closer control by the user of the final performance of the target system.

Thus, in summary, the invention provides a hardware/software codesign system for making an electronic circuit which includes both dedicated hardware and software controlled resources. The codesign system receives a behavioural description of the target electronic system and automatically partitions the required functionality between hardware and software, while being able to vary the parameters (e.g. size or power) of the hardware and/or software. Thus, for instance, the hardware and the processor for the software can be formed on an FPGA, each being no bigger than is necessary to form the desired functions. The codesign system outputs a description of the required processor or processors (which can be in the form of a net list for placement on the FPGA), machine code to run on the processor, and a net list or register transfer level description of the necessary hardware. It is possible for the user to write some parts of the description of the target system at register transfer level to give closer control over the operation of the target system, and the user can specify the processor or processors to be used, and can change, for instance, the partitioner, compilers or

speed estimators used in the codesign system. The automatic partitioning can be performed by using an optimisation algorithm, e.g. a genetic algorithm, which generates a partitioning based on estimates of performance.

The invention also allows the manual partition of systems across a number of hardware and software resources from a single behavioural description of the system. This provision for manual partitioning, as well as automatic partitioning, gives the system great flexibility.

The hardware resources may be a block that can implement random logic, such as an FPGA or ASIC; a fixed processor, such as a microcontroller, DSP, processor, or processor core; or a customizable processor which is to be implemented on one of the hardware resources, such as an FPGA-based processor. The system description can be augmented with register transfer level descriptions, and parameterised instantiations of both hardware and software library components written in other languages.

The sort of systems which can be targeted include:-

a fixed processor or processor core, coupled with custom hardware;

a set of customizable (e.g. FPGA-based) processors and custom hardware;

a system on a chip containing fixed processors and an FPGA; and

a PC containing an FPGA accelerator board.

The use of the advanced estimation techniques in specific embodiments of the invention allows the system to take into account the area of the processor that will be produced, allowing the targeting of customizable processors with additional and removable instructions, for example. The estimators also take into account the speed degradation produced when the logic that a fixed hardware resource must implement nears the resource's size limit. This is done by the estimator reducing the estimated speed as that limit is reached. Further, the estimators can operate on both the design before partitioning, and after partitioning. Thus high level simulation, as well as simulation and estimation after partitioning, can be performed.

Where the system is based on object oriented design, this allows the user to add new processors quickly and to easily define their compilers.

The part of the system which compiles the software can transparently support additional or absent instructions for the processor and so is compatible with the parametrization of the processor.

Preferably the input language supports variables with unspecified widths, which are then unified to a fixed width using a promotion scheme, and then mapped to the widths available on the target system architecture.

Further, in one embodiment of the invention, it is possible for the input description to include both behavioural and register transfer level descriptions, which can both be compiled to software. This gives support for very fast simulation and allows the user control of the behaviour of the hardware on each clock cycle.

The present invention will be further described by way of non-limitative example with reference to the accompanying drawings in which:

Figure 1 is a flow diagram schematically showing the codesign system of one embodiment of the invention;

Figure 2 illustrates the compiler objects which can be defined in one embodiment of the invention;

Figure 3 is a block diagram of the platform used to implement the second example circuit produced by an embodiment of the invention;

Figure 4 is a picture of the circuit of Figure 3;

Figure 5 is a block diagram of the system of Figure 3;

Figure 6 is a simulation of the display produced by the example of Figs. 3 to 5;

Figure 7 is a block diagram of a third example target system; and

Figure 8 is a block diagram showing a dependency graph for calculation of the variables in the Figure 7 example.

This description will later refer to specific examples of the input behavioural or register transfer level description of examples of target systems. These examples are reproduced in Appendices, namely:-

Appendix 1 is an example register transfer level description of a simple processor.

Appendix 2 is a register transfer level description of the main process flow in the example of Figs. 3 to 5.

Appendix 3 is the input specification for the target system of Fig. 8.

The flow of the codesign process in an embodiment of the invention is shown in Figure 1 and will be described below. The target architecture for this system is an FPGA containing one or more processors, and custom hardware. The processors may be of different architectures, and may communicate with each other and with the custom hardware.

The Input Language

In this embodiment the user writes a description 1 of the system in a C-like language, which is actually ANSI C with some additions which allow efficient compilation to hardware and parallel processes. This input description will be compiled by the system of Figure 1. The additions to the ANSI C language include the following:

Variables are declared with explicit bit widths and the operators working on the variables work with the required precision. This allows efficient implementation in hardware. For instance a statement which declares the width of variables (in this case the program counter pc, the instruction register ir, and the top of stack tos) is as follows :-

```
unsigned 12 pc, ir, tos
```

The width of the main data path of the processor in the target system may be declared, or else is calculated by the partitioner 7 as the width of the widest variable which it uses.

The "par" statement has been added to describe process-level parallelism. The system can automatically extract fine-grained parallelism from the C-like description but generating coarse-grained parallelism automatically is far more difficult. Consequently the invention provides this attribute to allow the user to express parallelism in the input language using the "par" statement which specifies that a following list of statements is to be executed in parallel. For example, the expression:-

```
par {
  parallel_port(port);
  SyncGen();
    }
```

means that two sub-routines, the first which is a driver for a parallel port and the second which is a sync generator for a video display are to be executed in parallel. All parts of the system will react to this appropriately.

Channels can be declared and are used for blocking, point-to-point synchronized communication as used in occam (see G. Jones. Programming in occam. Prentice Hall International Series in Computer Science, 1987, which is hereby incorporated by reference) with a syntax like a C function call. The parallel processes can use the channels to perform distributed assignment. Thus parallel processes can communicate using blocking channel communication. The keyword "chan" declares these channels. For example,

```
chan hwswchan;
```

declares a channel along which variables will be sent and received between the hardware and software parts of the system. Further,

```
send (channel_1, a)
```

is a statement which sends the value of variable a down channel_1; and

```
receive (channel_2, b)
```

is a statement which assigns the value received along channel_2 to variable b.

The hardware resources available are declared. The resources may be a customizable processor, a fixed processor, or custom hardware. The custom hardware may be a specific architecture, such as a Xilinx FPGA. Further, the architecture of the target system can be described in terms of the available functional units and their

interconnection.

To define the architecture "platforms" and "channels" are defined. A platform can be hard or soft. A hard platform is something that is fixed such as a Pentium processor or an FPGA. A soft platform is something that can be configured like an FPGA-based processor. The partitioner 7 understands the keywords "hard" and "soft", which are used for declaring these platforms and the codes can be implemented on any of these.

This particular embodiment supports the following hard platforms:

Xilinx 4000 series FPGAs (eg the Xilinx 4085 below);

Xilinx Virtex series FPGAs;

Altera Flex and APEX PLDs;

Processor architectures supported by ANSI C compilers;

and the following soft platforms each of which is associated with one of the parametrisable processors mentioned later:

FPGAStackProc, FPGAParallelStackProc, FPGAMips.

An attribute can be attached to a platform when it is declared:

platform (PLATFORMS)

For a hard platform the attribute PLATFORMS contains one element: the architecture of the hard platform. In this embodiment this may be the name of a Xilinx 3000 or 4000 series FPGA, an Altera FPGA, or an x86 processor.

For a soft platform, PLATFORMS is a pair. The first element is the architecture of the platform:-

FPGAStackProc, FPGAParallelStackProc or FPGAMips

and the second is the name of the previously declared platform on which the new platform is implemented.

Channels can be declared with an implementation, and as only being able to link previously declared platforms. The system 7 recognises the following channel implementations.

PCIBus - a channel implemented over a PCI bus between an FPGA card and a PC host.

FPGAChan - a channel implemented using wires on the FPGA.

The following are the attributes which can be attached to a channel when it is declared:

type(CHANNELTYPE)

This declares the implementation of the channel. Currently

CHANNELTYPE may be PCIBus or FPGAChan. FPGAChan is the default.

from(PLATFORM)

PLATFORM is the name of the platform which can send down the channel.

to(PLATFORM)

PLATFORM is the name of the platform which can receive from the channel.

The system 7 checks that the declared channels and the platforms that use them are compatible. The communication mechanisms which a given type of channel can implement are built into the system. New mechanisms can be added by the user, in a similar way to adding new processors as will be explained below.

Now an example of an architecture will be given.

Example Architecture

```
/* Architectural Declarations */
// the 4085 is a hard platform -- call this one meetea
board
hard meeteaBoard _attribute_ ((platform(Xilinx4085)));

// the pentium is a hard platform -- call this one
hostProcessor
hard hostProcessor _attribute_ ((platform(Pentium)));

// proc1 is a soft platform which is implemented
// on the FPGA on the meetea board
soft proc1 _attribute_ ((platform(FpgaStackProc,
meeteaBoard)));
```

Example Program

```
void main ( )
{
     // channel1 is implemented on a PCIBus
     // and can send data from hostProcessor to meetea
board
     chan channel1 _attribute_   ((type(PCIBus),
from (hostProcessor),
                                  to (meeteaBoard)));
     // channel2 is implemented on the FPGA
     chan channel2 _attribute_   ((type (FPGAChan)));

/* the code */
     par  {
               // code which can be assigned to
               // either hostProcessor (software),
               // or proc1 (software of reconfigurable
processor),
               // or meetea board (hardware),
```

```
                    // or left unassigned (compiler decides).

                    // Connections between hostProcessor
                    // and proc1 or meetea must be over the PCI
Bus
                    // (channel1)
                    // Connections between proc1 and hardware
                    // must be over the FPGA channel (channel2)
        }
}
```

Attributes are also added to the input code to enable the user to specify whether a block is to be put in hardware or software and for software the attribute also specifies the target processor. The attribute is the name of the target platform. For example:-

```
{
    int a, b;
    a = a + b;
} _attribute_ ((platform(hostProcessor)))
```

assigns the operation a + b to Host Processor.

For hardware the attribute also specifies whether the description is to be interpreted as a register transfer (RT) or behavioural level description. The default is behavioural. For example:-

```
{
    int a, b;
    par  {
        b = a + b;
        a = b;
    }
} _attribute_ ((platform(meeteaBoard), level (RTL)))
```

would be compiled to hardware using the RTL compiler, which would guarantee that the two assignments happened on the same clock cycle.

Thus parts of the description which are to be allocated to hardware can be written by the user at a register transfer level, by using a version of the input language with a well defined timing semantics (for example Handel-C or another RTL language), or the scheduling decisions (i.e. which operations happen on which clock cycle) can be left to the compiler. Thus using these attributes a block of code may be specifically assigned by the user to one of the available resources. Soft resources may themselves be assigned to hardware resources such as an FPGA-based processor. The following are the attributes which can be attached to a block of code:

platform(PLATFORM)

PLATFORM is the name of the platform on which the code will be implemented. This in conjunction with the level () attribute (see below) implies the compiler which will be used to compile that code.

level(LEVEL)

LEVEL is Behavioural or RTL. Behavioural description will be scheduled and may be partitioned. RTL descriptions are passed straight through the RTL synthesiser e.g. a Handel-C compiler.

cycles(NUMBER)

NUMBER is a positive integer. Behavioural descriptions will be scheduled in such a way that the block of code will execute within that number of cycles, when the compiler is able. An error is generated if it is not possible.

For partitioning of programs which include pointers, the concept of address space partitioning is introduced. This is to solve the problems of inefficiency or implementation which can arise with pointers which point to locations which are partitioned to different physical memories. For instance, some pointers may point to

locations in memory on hardware, and some may point to locations in RAM used by a processor. Such partitioning can result in, for example, excessive data transfers being necessary between the different memories. To solve these problems, on compilation every variable and function in the program is assigned its own address space. The compiler locates variables and functions which, because of operations between them, must share the same address space, and it then unifies these address spaces. If as a result of manual partitioning a unified address space is split across a plurality of physical memories, accesses to the address spaces and communications between them are synthesised by the compiler as appropriate. During manual partitioning of the system the user can use the standard C "memcpy" function to pass blocks of data between address spaces to improve the efficiency of the program by reducing the number of individual data transfers.

Thus the use of this input language which is based on a known computer language, in this case C, but with the additions above allows the user, who could be a system programmer, to write a specification of the system in familiar behavioural terms like a computer program. The user only needs to learn the additions above, such as how to declare parallelism and to declare the available resources to be able to write the input description of the target system.

This input language is input to the parser 3 which parses and type checks the input code, and performs some syntax level optimizations, (in a standard way for parsers), and attaches a specific compiler to the appropriate blocks of code based on the attributes above. The parser 3 uses standard techniques [Aho, Sethi and Ullman; "Compilers Principles, Techniques, and Tools"; Addison Wesley known as "The Dragon Book", which is hereby incorporated by reference] to turn the system description in the input language into an internal data structure, the abstract syntax tree which can be supplied to the partitioner 7.

The width adjuster 5 uses C-like techniques to promote automatically the arguments of operators to wider widths such that they are all of the same width for instance by concatenating them with zeros. Thus this is an extension of the promotion scheme of the C language, but uses arbitrary numbers of bits. Further adjustment is carried out later in the flow at 5a and 5b, for instance by ANDing them

with a bit mask. Each resources has a list of widths that it can support. For example a 32 bit processor may be able to carry out 8, 16 and 32 bit operations. Hardware may be able to support any width, or a fixed width datapath operator may have been instantiated from a library. The later width adjustment modules 5a and 5b insert commands to enable the width of operation in the description to be implemented correctly using the resources available.

Hardware/Software Partitioning

The partitioner 7 generates a control/data-flow graph (CDFG) from the abstract syntax tree, for instance using the techniques described in G. de Micheli "Synthesis and Optimization of Digital Circuits"; McGraw-Hill, 1994 which is hereby incorporated by reference. It then operates on the parts of the description which have not already been assigned to resources by the user. It groups parts of the description together into blocks, "partitioning blocks", which are indivisible by the partitioner. The size of these blocks is set by the user, and can be any size between a single operator, and a top-level process. Small blocks tend to lead to a better partition which takes longer to generate; larger blocks tend to lead to a worse partition which is generated more quickly.

The algorithm used in this embodiment is described below but the system is designed so that new partitioning algorithms can easily be added, and the user can choose which of these partitioning algorithms to use. The algorithms all assign each partitioning block to one of the hardware resources which has been declared. The algorithms do this assignment so that the total estimated hardware area is no larger than the hardware resources available, and so that the estimated speed of the system is maximised.

The algorithm implemented in this embodiment of the system is a genetic algorithm for instance as explained in D.E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine learning", Addison-Wesley, 1989 which is hereby incorporated by reference. The resource on which each partitioning block is to be placed is represented by a gene. Additional genes represent possible parametrisations

of customizable processors. The fitness function returns the estimated system speed multiplied by a factor k, $0 < k \leq 1$; $k = 1$ if estimators say the partitioning will fit the available hardware; $k < 1$ otherwise, becoming rapidly less favourable as size increases. Different partitions are generated and estimated speed found. The user may set the termination condition to one of the following:-

1) when the estimated system speed meets a given constraint;
2) when the result converges, i.e. the algorithm has not resulted in improvement after a user-specified number of iterations;
3) when the user terminates the optimisation manually.

The partitioner 7 uses estimators 19, 21, and 23 to estimate the size and speed of the hardware, software and interfaces as described below.

It should be noted from Figure 1 that the estimators and the simulation and profiling module 19 can accept a system description from several levels in the flow. Thus it is possible for the input description, which may include behavioural and register transfer level parts, to be compiled to software for simulation and estimation at this stage. Further, the simulator can be used to collect profiling information for sets of typical input data, which will be used by the partitioner 7 to estimate data dependent values, by inserting data gathering operations into the output code.

Hardware Estimation

The estimator 21 is called by the partitioner 7 for a quick estimation of the size and speed of the hardware parts of the system using each partition being considered. Data dependent values are estimated using the average of the vales for the sets of typical input data supplied by the user.

To estimate the speed of hardware, the description is scheduled using a call to the behavioural synthesiser 11. The user can choose which estimation algorithm to use, which gives a choice between slow accurate estimation and faster less accurate estimation. The speed and area of the resulting RTL level description is then estimated

using standard techniques. For FPGAs the estimtate of the speed is then decreased by a non-linear factor determined from the available free area, to take into account the slower speed of FPGA designs when the FPGA is nearly full.

Software Estimation

If the software is to be implemented on a fixed processor, then its speed is estimated using the techniques described in J. Madsen and J. Grode and P.V. Knudsen and M.E. Petersen and A. Haxthausen, "LYCOS: the Lyngby Co-Synthesis System, Design Automation of Embedded Systems, 1977, volume 2, number 2, (Madsen et al) which is hereby incorporated by reference. The area of software to be implemented on a fixed processor is zero.

If the target is customizable processors to be compiled by the system itself then a more accurate estimation of the software speed is used which models the optimizations that the software compiler 15 uses. The area and cycle time of the processor is modelled using a function which is written for each processor, and expresses the required values in terms of the values of the processor's parametrizations, such as the set of instructions that will be used, the data path and instruction register width and the cache size.

Interface Synthesis and Estimation

Interfaces between the hardware and software are instantiated by the interface cosynthesizer 9 from a standard library of available communication mechanisms. Each communication mechanism is associated with an estimation function, which is used by the partitioner to cost the software and hardware speed and area required for given communication, or set of communications. Interfaces which are to be implemented using a resource which can be parametrised (such as a channel on an FPGA), are synthesised using the parametrizations decided by the partitioner. For example, if a transfer of ten thousand 32 bit values over a PCI bus was required, a DMA transfer from the host to an FPGA card's local memory might be used.

Compilation

The compiler parts of the system may be designed in an object oriented way, providing a class hierarchy of compilers, as shown for example in Figure 2. Each node in the tree shows a class which is a subclass of its parent node. The top-level compiler class provides methods common to both the hardware and software flows, such as the type checking, and a system-level simulator used for compiling and simulating the high-level description. These methods are inherited by the hardware and software compilers, and may be used or overridden. The compiler class also specifies other, virtual, functions which must be supplied by its subclass. So the compile method on the hardware compiler class compiles the description to hardware by converting the input description to an RTL description; the compile method on the Processor A compiler compiles a description to machine code which can run on Processor A.

There are two ways in which a specific compiler can be attached to a specific block of code:

A) In command line mode. The compiler is called from the command line by the attributes mentioned above specifying which compiler to use for a block of code.

B) Interactively. An interactive environment is provided, where the user has access to a set of functions which the user can call, e.g. to estimate speed and size of hardware and software implementations, manually attach a compiler to a block code, and call the simulator. This interactive environment also allows complex scripts, functions and macros to be written and saved by the user for instance so that the user can add a new partitioning algorithm..

The main compilation stages of the process flow are software or hardware specific. Basically at module 11 the system schedules and allocates any behavioural parts of the hardware description, and at module 15 compiles the software description to assembly code. At module 17 it also writes a parametrised description of the

processors to be used, which may also have been designed by the user. These individual steps will be explained in more detail.

Hardware Compilation

The parts of the description to be compiled into hardware use a behavioural synthesis compiler 11 using the techniques of De Micheli mentioned above. The description is translated to a control/data flow graph, scheduled (i.e. what happens on each clock cycle is established) and bound (i.e. which resources are used for which operations is established), optimised, and then an RT-level description is produced.

Many designers want to have more control over the timing characteristics of their hardware implementation. Consequently the invention also allows the designer to write parts of the input description corresponding to certain hardware at the register transfer level, and so define the cycle-by-cycle behaviour of that hardware.

This is done by using a known RT-level description with a well-defined timing semantics such as Handel-C. In such a description each assignment takes one clock cycle to execute, control structures add only combinational delay, and communications take one clock cycle as soon as both processes are ready. With the invention an extra statement is added to this RT-level version of the language: "delay" is a statement which uses one clock cycle but has no other effect. Further, the "par" attribute may again be used to specify statements which should be executed in parallel.

Writing the description at this level, together with the ability to define constraints for the longest combinational path in the circuit, gives the designer close control of the timing characteristics of the circuit when this is necessary. It allows, for example, closer reasoning about the correctness of programs where parallel processes write to the same variable. This extra control has a price: the program must be refined from the more general C description, and the programmer is responsible for thinking about what the program is doing on a cycle-by-cycle basis. An example of a description of a processor at this level will be discussed later.

The result of the hardware compilation by the behavioural synthesiser 11 is an RTL description which can be output to a RTL synthesis system 13 using a hardware

description language (e.g. Handel-C or VHDL), or else synthesized to a gate level description using the techniques of De Micheli.

RTL synthesis optimizes the hardware description, and maps it to a given technology. This is performed using standard techniques.

Software Compilation

The software compiler 15 largely uses standard techniques [e.g. from Aho, Sethi and Ullman mentioned above]. In addition, parallelism is supported by mapping the invention's CSP-like model of parallelism and communication primitives into the target model. For instance channels can mapped to blocks of shared memory protected by semaphores. CSP is described in C.A.R. Hoare "Communicating sequential processes." Prentice-Hall International Series in Computing Science. Prentice-Hall International, Englewood Cliffs, N.J. which is hereby incorporated by reference.

Compound operations which are not supported directly by the processor are decomposed into their constituent parts, or mapped to operations on libraries. For example multiply can be decomposed into shifts and adds. Greedy pattern matching is then used to map simple operations into any more complex instructions which are supported by the processor. Software can also be compiled to standard ANSI C, which can then be compiled using a standard compiler. Parallelism is supported by mapping the model in the input language to the model of parallelism supported by the C compiler, libraries and operating system being used.

The software compiler as exemplified in Figure 2 is organised in an object oriented way to allow users to add support for different processors (see Figure 2) and for processor parametrisations. For example, in the processor parametriser 17 unused instructions from the processor description are automatically removed, and support for additional instructions can be added. This embodiment of the invention, includes some pre-written processor descriptions which can be selected by the user. It contains parametrised descriptions of three processors, and the software architecture is designed so that it is easy for developers to add new descriptions which can be completely new

or refinements of these. The three processors provided are:-

A Mips-like processor, similar to that described in [Patterson and Hennessy, Computer Organisation and Design, 2nd Edition, Morgan Kauffman].

A 2-cycle non-pipeline stack-based processor (see below).

A more sophisticated multicycle non-pipelined stack-based processor, with a variable number of cycles per instruction, and hardware support for parallelism and channels.

Thus the software compiler supports many processor parametrisations. More complex and unexpected modifications are supported by virtue of the design of the compiler (e.g. in an object oriented way as above), which allows small additions to be made easily by the user. Most of the mapping functions can be inherited from existing processor objects, minor additions can be made and a function used to calculate the speed and area of processor given the parametrizations of the processor and a given program. The output of the software compilation/processor parametrization process is machine code to run on the processor together with a description of the processor to be used (if it is not a standard one).

Co-Simulation and Estimation

The scheduled hardware, register transfer level hardware, software and processor descriptions are then combined. This allows a cycle-accurate co-simulation to be carried out, e.g. using the known Handel-C simulator, though a standard VHDL or Verilog simulator and compiler could be used.

Handel-C provides estimation of the speed and area of the design, which is written as an HTML file to be viewed using a standard broswer, such as Netscape. The file shows two versions of the program: in one each statement is coloured according to how much area it occupies, and in the other according to how much

combinational delay it generates. The brighter the colour for each statement, the greater the area or delay. This provides a quick visual feedback to the user of the consequences of the design decisions.

The Handel-C simulator is a fast cycle-accurate simulator which uses the C-like nature of the specification to produce an executable file which simulates the design. It has an X-windows interface which allows the user to view VGA video output at about one frame per second.

When the user is happy with the RT-level simulation and the design estimates then the design can be compiled to a netlist. This is then mapped, placed and routed using the FPGA vendor's tools.

The simulator can be used to collect profiling information for sets of typical input data, which will be used by the partitioner 7 to estimate data dependent values, by inserting data gathering operations into the output code. For example the source program can be compiled to platform independent bytecode. A suitable bytecode interpreter is then augmented such that accesses to memory (typically load and store instructions) can be traced. In this way the memory use behaviour of each part of the source program can be examined by executing the program and analysing the generated trace.

However, a simplistic implementation of this technique suffers from the problem of generating a very large amount of profiling data. There are two alternative techniques to solve this problem:

1. During execution of a single function (or set of functions grouped as a domain) a map of all the memory accessed is recorded. At the end of execution of the function only a compressed version of this map (compressed using a technique such as run-length encoding) is output. Since functions will typically tend to use blocks of memory in ranges, rather than a fully random access pattern, this results in significant savings in the size of the generated output. The output is then analysed post-hoc to determine where memory transfers would have taken place between domains of a partitioned system.

2. Alternatively, some of the analysis can happen on-line during the execution of the program. In this case, a memory map is kept of the program which

records which functions (or groups of functions) have valid copies of small ranges of memory (micropages). When a function reads for an area of memory, this map is checked to see which functions have a valid copy of the data. If the current function was a valid copy no further action is taken. If no function has a valid copy of the data then it is taken as coming from an external source function. Otherwise a transfer from one of the other functions to the current function is recorded, and the map records that the current function now has a valid copy of the micropage. When a write occurs, exactly the same action takes place except the ownership of the micropage becomes only the current function, no other functions now possess valid (up-to-date) copies of the data in the given page. The result of the execution of a program in this way is a 2-dimensional table recording data transfers from functions to functions. This data can then be further analysed to give estimates for the performance of given partitions, be used to decide partitions, or be presented in a graphical form (such as a directed graph).

Implementation Language

The above embodiment of the system was written in objective CAML which is a strongly typed functional programming language which is a version of ML but obviously it could be written in other languages such as C.

Provable Correctness

A subset of the above system could be used to provide a provably correct compilation strategy. This subset would include the channel communication and parallelism of OCCAM and CSP. A formal semantics of the language could be used together with a set of transformations and a mathematician, to develop a provably correct partitioning and compilation route.

Some examples of target system designed using the invention will be now described.

EXAMPLE 1 - PROCESSOR DESIGN

The description of the processor to be used to run the software part of the target system may itself be written in the C-like input language and compiled using the codesign system. As it is such an important element of the final design most users will want to write it at the register transfer level, in order to hand-craft important parts of the design. Alternatively the user may use the predefined processors, provided by the codesign system or write the description in VHDL or even at gate level, and merge it into the design using the FPGA vendor's tools.

With this system the user can parametrise the processor design in nearly any way that he or she wishes as discussed above in connection with the software compilation and as detailed below.

The first processor parametrisation to consider is removing redundant logic. Unused instructions can be removed, along with unused resources, such as the floating point unit or expression stack.

The second parametrisation is to add resources. Extra RAMs and ROMs can be added. The instruction set can be extended from user assigned instruction definitions. Power-on bootstrap facilities can be added.

The third parametrisation is to tune the size of the used resources. The bit widths of the program counter, stack pointer, general registers and the opcode and operand portions of the instruction register can be set. The size of internal memory and of the stack or stacks can be set, the number and priorities of interrupts can be defined, and channels needed to communicate with external resources can be added. This freedom to add communication channels is a great benefit of codesign using a parametrisable processor, as the bandwidth between hardware and software can be changed to suit the application and hardware/software partitioning.

Finally, the assignment of opcodes can be made, and instruction decoding modified accordingly.

The user may think of other parametrisations, and the object oriented processor description allows this.

The description of a very simple stack-based processor in this style (which is actually one of the pre-written processors provided by the codesign system for use by

the user) is listed in Appendix 1.

Referring to Appendix 1, the processor starts with a definition of the instruction width, and the width of the internal memory and stack addresses. This is followed by an assignment of the processor opcodes. Next the registers are defined; the declaration "unsigned x y, z" declares unsigned integers y and z of width x. The program counter, instruction register and top-of-stack are the instruction width; the stack pointer is the width of the stack's address bus.

After these declarations the processor is defined. This is a simple non-pipelined two-cycle processor. On the first cycle (the first three-line "par"), the next instruction is fetched from memory, the program counter is incremented, and the top of the stack is saved. On the second cycle the instruction is decoded and executed. In this simple example a big switch statement selects the fragment of code which is to be executed.

This simple example illustrates a number of points. Various parameters, such as the width of registers and the depth of the stack can be set. Instructions can be added by including extra cases in the switch statement. Unused instructions and resources can be deleted, and opcodes can be assigned.

The example also introduces a few other features of the register transfer level language such as rom and ram declarations.

EXAMPLE 2 - VIDEO GAME

To illustrate the use of the invention using an application which is small enough to describe easily a simple Internet video game was designed. The target system is a video game in which the user can fly a plane over a detailed background picture. Another user can be dialled up, and the screen shows both the local plane and a plane controlled remotely by the other user. The main challenge for the design is that the system must be implemented on a single medium-sized FPGA.

Implementation Platform

The platform for this application was a generic and simple FPGA-based board.

A block diagram of the board, a Hammond board, is shown in Figure 3, and a picture is shown in Figure 4.

The Hammond board contains a Xilinx 4000 series FPGA and 256kb synchronous static RAM. Three buttons provide a simple input device to control the plane; alternatively a standard computer keyboard can be plugged into the board. There is a parallel port which is used to configure the FPGA, and a serial port. The board can be clocked at 20 MHz from a crystal, or from a PLL controlled by the FPGA. Three groups of four pins of the FPGA are connected to a resistor network which gives a simple digital to analogue converter, which can be used to provide 12 bit VGA video by implementing a suitable sync generator on the FPGA.

Problem Description and Discussion

The specification of the video game system is as follows:

The system must dial up an Internet service provider, and establish a connection with the remote game which will be running on a workstation.

The system must display a reconfigurable background picture.

The system must display on a VGA monitor a picture of two planes: the local plane and the remote plane.

The position of the local plane will be controlled by the buttons on the Hammond board.

The position of the remote plane will be received over the dialup connection every time it changes.

The position of the local plane will be sent over the dialup connection

every time it changes.

This simple problem combines some hard timing constraints, such as sending a stream of video to the monitor, with some complex tasks without timing constraints, such as connecting to the Internet service provider. There is also an illustration of contention for a shared resource, which will be discussed later.

System Design

A block diagram of the system is shown in Figure 5. The system design decisions were quite straightforward. A VGA monitor is plugged straight into the Hammond board. To avoid the need to make an electrical connection to the telephone network a modem was used, and plugged into the serial port of the Hammond board. Otherwise it would have been quite feasible to build a simple modem in the FPGA.

The subsystems required are:

serial port interface,
dial up,
establishing the network connection,
sending the position of the local plane,
receiving the position of the remote plane,
displaying the background picture,
displaying the planes.

A simple way of generating the video is to build a sync generator in the FPGA, and calculate and output each pixel of VGA video at the pixel rate. The background picture can be stored in a "picture RAM". The planes can be stored as a set of 8x8 characters in a "character generator ROM", and the contents of each of the characters' positions on the screen stored in a "character location RAM".

Hardware/software partitioning

The hardware portions of the design are dictated by the need of some parts of the system to meet tight timing constraints. These are the video generation circuitry and the port drivers. Consequently these were allocated to hardware, and their C descriptions written at register transfer level to enable them to meet the timing constraints. The picture RAM and the character generator ROM and character location RAM were all stored in the Hammond board RAM bank as the size estimators showed that there would be insufficient space on the FPGA.

The parts of the design to be implemented in software are the dial-up and negotiation, establishing the network, and communicating the plane locations. These are non-time critical, and so can be mapped to software. The program is stored in the RAM bank, as there is not space for the application code in the FPGA.

The main function is shown in Appendix 2. The first two lines declare some communication channels. Then the driver for the parallel port and sync generator are started, and the RAM is initialised with the background picture, the character memory and the program memory. The parallel communicating hardware and software process are then started, communicating over a channel hwswchan. The software establishes the network connection, and then enters a loop which transmits and receives the position of the local and remote plane, and sends new positions to the display process.

Processor Design

The simple stack-based processor from Appendix 1 was parametrised in the following ways to run this software. The width of the processor was made to be 10 bits, which is sufficient to address a character on the screen in a single word. No interrupts were required, so these were removed, as were a number of unused instructions, and the internal memory.

Co-Simulation

The RT-level design was simulated using the Handel-C simulator. Sample input files mimicking the expected inputs from the peripherals were prepared, and these were fed into the simulator. A black and white picture of the colour display is shown in Figure 6 (taken as a snapshot of the X window drawn by the co-simulator).

The design was then placed and routed using the proprietary Xilinx tools, and successfully fit into the Xilinx 4013 FPGA on the Hammond board.

This application would not have been easy to implement without the codesign system of the invention. A hardware-only solution would not have fitted onto the FPGA; a software-only solution would not have been able to generate the video and interface with the ports at the required speed. The invention allows the functionality of the target system to be partitioned while parametrizing the processor to provide an optimal system.

Real World Complications

The codesign system was presented with an implementation challenge with this design. The processor had to access the RAM (because that is where the program was stored), whilst the hardware display process simultaneously had to access the RAM because this is where the background picture, character map and screen map were stored. This memory contention problem was made more difficult to overcome because of an implementation decision made during the design of the Hammond board: for a read cycle the synchronous static RAM which was used requires the address to be presented the cycle before the data is returned.

The display process needs to be able to access the memory without delay, because of the tight timing constraints placed on it. A semaphore is used to indicate when the display process requires the memory. In this case the processor stalls until the semaphore is lowered. On the next cycle the processor then presents to the memory the address of the next instruction, which in some cases may already have been presented once.

The designer was able to overcome this problem using the codesign system of invention because of the facility for some manual partitioning by the user and describing some parts of the design at the register transfer level to give close control over those parts. Thus while assisting the user, the system allows close control where desired.

EXAMPLE 3 - MASS-SPRING SIMULATION

Introduction

The "springs" programme is a small example of a codesign programmed in the C-like language mentioned above. It performs a simulation of a simple mass-spring system, with a real time display on a monitor, and interaction via a pair of buttons.

Design

The design consists of three parts: a process computing the motion of the masses, a process rendering the positions of the masses into line segments, and a process which displays these segments and supplies the monitor with appropriate synchronisation signals. The first two processes are written in a single C-like program. The display process is hard real-time and so requires a language which can control external signals at the resolution of a single clock cycle, so for this reason it is implemented using an RTL description (Handel-C in this instance). These two programs are shown in appendix 3. They will be explained below, together with the partitioning process and the resulting implementation. Figure 7 is a block diagram of the ultimate implementation, together with a representation of the display of the masses and springs. Figure 8 is a dependency graph for calculation of the variables required.

Mass motion process

The mass motion process first sets up the initial positions, velocities and acceleration of the masses. This can be seen in appendix 3 where positions p0 to p7 are initialised as 65536. The program then continues in an infinite loop, consisting of: sending pairs of mass positions to the rendering process, computing updated positions based on the velocities of the masses, computing updated velocities based on the accelerations of the masses, and computing accelerations based on the positions of the masses according to Hooke's law. The process then reads the status of the control buttons and sets the position of one of the masses accordingly. This can be seen in appendix 3 as the statement "received (buttons, button_status);".

This process is quite compute intensive over a short -period (requiring quite a number of operations to perform the motion calculation), but since these only occur once per frame of video the amortised time available for the calculation is quite long.

Rendering process

The rendering process runs an infinite loop performing the following operations: reading a pair of mass positions from the mass motion process then interpolating in between these two positions for the next 64 lines of video output. A pair of interpolated positions is sent to the RTL display process once per line. This is a relatively simple process with only one calculation, but this must be performed very regularly.

Display Process

The display process (which is written in Handel-C) and is illustrated in appendix 3 reads start and end positions from the rendering process and drives the video colour signal between these positions on a scan line. Simultaneously, it drives the synchronisation signals for the monitor. At the end of each frame it reads the values from the external buttons and sends these to the mass motion process.

Partitioning by the codesign system

The design could be partitioned in a large number of ways. The system could partition the entire design into hardware or into software, partition the design at a high level by the first two processes described above or it can partition the design at a lower level and generate further parallel processes communicating with each other. Whatever choice the partitioner makes, it maintains the functional correctness of the design, but will change the cost of the implementation (in terms of the area, clock cycles and so forth). The user may direct the partitioner to choose one of the options in preference to the others. A number of the options are described below.

Pure hardware

The partitioner could map the first two processes directly into Handel-C, after performing some additional parallelisation. The problem with this approach is that each one of the operations in the mass motion process will be dedicated to its own piece of hardware, in an effort to increase performance. However, as discussed above, this is unnecessary as these calculations can be performed at a slower speed. The result is a design that can perform quickly enough but which is too large to fit on a single FPGA. This problem would be recognised by the partitioner using its area estimation techniques.

Pure software

An alternative approach is for the partitioner to map the two processes into software running on a parametrised threaded processor. This reduces the area required, since the repeated operations of the mass motion calculations are performed with a single operation inside the processor. However, since the processor must swap between doing the mass motion calculations and the rendering calculations, overhead is introduced which causes it to run too slowly to display in real-time. The partitioner can recognise this by using the speed estimator, based on the profiling information gathered from simulations of the system.

Software/software

Another alternative would be for the partitioner to generate a pair of parametrised processors running in parallel, the first calculating motion and the second performing the rendering. The area required is still smaller than the pure hardware approach, and the speed is now sufficient to implement the system in real time. However, using a parametrised processor for the rendering process adds some overhead (for instance, performing the instruction decoding), which is unnecessary. So although the solution works, it is sub-optimal.

Hardware/Software

The best solution, and the one chosen by the partitioner, is to partition the mass motion process into software for a parametrised, unthreaded processor, and to partition the rendering process 74 which was written at a behavioural level together with the position, velocity and acceleration calculations 72 into hardware. This solution has the minimum area of the options considered, and performs sufficiently quickly to satisfy the real time display process.

Thus referring to Figure 7, the behavioural part of the system 70 includes the calculation of the positions, velocities and accelerations of the masses at 72 (which will subsequently be partitioned to software), and the line and drawing processes at 74 (which will subsequently be partitioned to hardware). The RTL hardware 80 is used to receive the input from the buttons at 82 and output the video at 84.

Thus the partitioner 7 used the estimators 19, 21 and 23 to estimate the speed and area of each possible partition based on the use of a customised processor. The interface cosynthesiser 9 implements the interface between hardware and software on two FPGA channels 71 and 73 and these are used to transfer a position information to the rendering process and to transfer the button information to the position calculation 72 from button input 82.

The width adjuster 5, which is working on the mass motion part of the problem to be partitioned to software, parametrises the processor to have a width of

17 bits and adjusts the width of "curr_pos" which is the current position to nine bits, the width of the segment channel. The processor parametriser at 17 further parametrises the processor by removing unused instructions such as multiply, interrupts, and the data memory is reduced and multi-threading is removed. Further, op codes are assigned and the operator width is adjusted.

The description of the video output 84 and button interface 82 were, in this case, written in an RTL language, so there is no behavioural synthesis to be done for them. Further, because the hardware will be formed on an FPGA, no width adjustment is necessary because the width can be set as desired.

The partitioner 7 generates a dependency graph as shown in Figure 8 which indicates which variables depend on which. It is used by the partitioner to determine the communications costs associated with the partitioning, for instance to assess the need for variables to be passed from one resource to another given a particular partitioning.

SUMMARY

Thus the codesign system of the invention has the following advantages in designing a target system:-

1. It uses parametrisation and instruction addition and removal for processor design in an FPGA tailored to the application. The system provides an environment in which an FPGA-based processor and its compiler can be developed in a single framework.

2. It can generate designs containing multiple communicating processors, parametrised custom processors, and the inter-processor communication can be tuned for the application.

3. The hardware can be designed to run in parallel with the processors to meet speed constraints. Thus time critical parts of the system can be allocated to custom hardware, which can be designed at the behavioural or register transfer level.

4. Non-time critical parts of the design can be allocated to software, and run on a small, slow processor.

5. The system can target circuitry on dynamic FPGAs. The FPGA can contain a small processor which can configure and reconfigure the rest of the FPGA at run time.

6. The system allows the user to explore efficient system implementations, by allowing parametrised application-specific processors with user-defined instructions to communicate with custom hardware. This combination of custom processor and custom hardware allows a very large design space to be explored by the user.

Appendix 1

```
Register transfer level description of simple processor

void sw ( )
{
#define iw = 12;              /* instruction
                                 width */
#define mw = 3; /* memory address width */
#define CONST = 0             /* push constant */
#define LOAD = 1              /* push variable */
#define GLOBAL = 2            /* push address */
#define PUTCHAR = 15          /* put a character
                                 along the
                                 standard output
                                 channel */
#define GETCHAR = 16          /* get a character
                                 from the standard
                                 input channel */

...

   rom program [ ] = {
#include "prog.o"
    };
    ram stack [1<<mw] with { dualport = 1 };
    ram memory [1<<mw];
    unsigned iw pc, ir, tos;
    unsigned mw sp;

    do   {
      par   {
          ir = program[pc];
          pc = pc + 1;
          tos = stack[sp-1]; /* save top of
                                stack to avoid
                                two ram accesses
                                in one cycle
                             /*
    }
    switch (ir)   {
    case CONST :
      par   {
          stack [sp] = program[pc];
          sp = sp+1;
          pc = pc+1;
       }
      break;
    case  LOAD :
```

```
                stack[sp-1] = memory[tos<-mw];
                break;

            ...

            case  STOP :
              break;
            default : /* unknown opcode */
              while (1) delay;
            }
        } while (ir ! = STOP);
    }
```

Appendix 2

RTL description of main

```
void main ( )
{
   chan hwswchan;
   Chan unsigned 8 port;

   par  {
       parallel_port (port);
       SyncGen ( );
       {
          initialiseRam (port);
          par  {
             display (hwswchan);
              sw (hwswchan);
          }
       }
   }
}
```

Appendix 3

CALCULATION PROCESS

```
/*
  * Channel communicating object positions
  */
  chan unsigned 17 position;

/*
  * Channel communicating segment information
  */
```

```
    chanout unsigned 9 segment;

  /*
    * Channel communicating button information
    */
    chanin unsigned 2 buttons;

  /*
    * Overall par
    */
  par
  {
     /*
       * Mass motion
       */
     {
         /*
           * Positions of each mass, 9+8 fixed point
           */
        unsigned 17 p0, p1, p2, p3, p4, p5, p6, p7;

          /*
            * Velocity of each mass, 9+8 fixed point
            */
          int 17 v1, v2, v3, v4, v5, v6, v7;

          /*
            * Accelerations of each mass, 9+8 fixed point
            */
          int 17 a1, a2, a3, a4, a5, a6, a7;

          /*
            * Button status
            */
          unsigned 2 button_status;

          /*
            * Initial setup of positions
            */
          p0 = 65536;
          p1 = 65536;
          p2 = 65536;
          p3 = 65536;
          p4 = 65536;
          p5 = 65536;
          p6 = 65536;
          p7 = 65536;

  /*
    * Forever. . .
    */
  while  (1)
       {
```

```
        /*
          * Send successive positions down position channel
          */
        send (position, p0);
        send (position, p1);
        send (position, p1);
        send (position, p2);
        send (position, p2);
        send (position, p3);
        send (position, p3);
        send (position, p4);
        send (position, p4);
        send (position, p5);
        send (position, p5);
        send (position, p6);
        send (position, p6);
        send (position, p7);

        /*
          * Update positions according to velocities
          */
        p1  +=  (unsigned 17) v1;
        p2  +=  (unsigned 17) v2;
        p3  +=  (unsigned 17) v3;
        p4  +=  (unsigned 17) v4;
        p5  +=  (unsigned 17) v5;
        p6  +=  (unsigned 17) v6;
        p7  +=  (unsigned 17) v7;

        /*
          * Update velocities according to accelerations
          */
        v1  +=  a1  -  (v1  >>  6);
        v2  +=  a2  -  (v2  >>  6);
        v3  +=  a3  -  (v3  >>  6);
        v4  +=  a4  -  (v4  >>  6);
        v5  +=  a5  -  (v5  >>  6);
        v6  +=  a6  -  (v6  >>  6);
        v7  +=  a7  -  (v7  >>  6);

        /*
          * Set accelerations according to relative positions
          */
        a1  =  (int  17)  ( ( ( p2  >>  8 )  -  ( p1  >>  8 ) )  +
( ( p0  >>  8 )  -  ( p1  >>  8 ) ) );
        a2  =  (int  17)  ( ( ( p3  >>  8 )  -  ( p2  >>  8 ) )  +
( ( p1  >>  8 )  -  ( p2  >>  8 ) ) );
        a3  =  (int  17)  ( ( ( p4  >>  8 )  -  ( p3  >>  8 ) )  +
( ( p2  >>  8 )  -  ( p3  >>  8 ) ) );
        a4  =  (int  17)  ( ( ( p5  >>  8 )  -  ( p4  >>  8 ) )  +
```

```
( ( p3 >> 8 ) - ( p4 >> 8 ) ) );
        a5 = (int 17) ( ( ( p6 >> 8 ) - ( p5 >> 8 ) ) +
( ( p4 >> 8 ) - ( p5 >> 8 ) ) );
        a6 = (int 17) ( ( ( p7 >> 8 ) - ( p6 >> 8 ) ) +
( ( p5 >> 8 ) - ( p6 >> 8 ) ) );
        a7 = (int 17) ( ( p6 >> 8) - ( p7 >> 8) );


        /*
          *    Get button information
          */
        receive (buttons, button_status);

        /*
          *    Fix top point according to buttons
          */
        if    (button_status & 1)
            {
                p0 = 65536 - 16384;
            }
        else if (button_status & 2)
            {
                p0 = 65536 + 16384;
            }
        else
            {
                p0 = 65536;
            }
        }
    }

    /*
      * Line drawing
      */
    {
       /*
         * Positions of previous and next masses positions
         */
    unsigned 17  prev_pos,  next_pos,  curr_pos;

    /*
      * Which line of interpolation
      */
    unsigned char line;

    /*
      * Forever . . .
      */
    while (1)
        {
```

```
        /*
          * Receive previous mass position
          */
        receive (position, prev_pos);
        curr_pos = prev_pos;

    /*
      * Read next mass position
      */
    receive (position, next_pos);

    /*
      * Do 64 lines of interpolation
      */
    for (line = 0; line != 64; line++)
        {
          /*
            * Send start position of segment
            */
          send (segment, curr_pos >> 8); /**width adjustment : 17
    along
                  channel of
                    width 9 so takes bottom 9 bits */

                /*
                  * Move by appropriate amount (1/64 total change)
                  */
                curr_pos += (unsigned 17)   ( ( (int 17) next_pos -
                                            (int 17) prev_pos) >> 6);

                /*
                  * Send end position of segment
                  */
                send (segment, curr_pos >> 8);
            }
        }
    }
}

DISPLAY PROCESS


/* standard includes */
#include   "hammond.h"
#include   "syncgen.h"
#include   "stdlib.h"
#include   "parallel.h"

/*
  * Segment information channel
  */
chan segment;
```

```
/*
  * Button information channel
  */
chan buttons

/*
  * Include dash generated stuff
  */
#include   "Handelc.h"

/*
  * Main program
  */
void main ( )
{
    /*
      * Scan positions
      */
    unsigned sx, sy;

/*
  * Video output register
  */
unsigned 1 video;

/*
  * Video output bus
  */
interface bus_out ( )   video out (Visible (sx,  sy)   ?
                                   (video   ?   (unsigned 12)
0xfff   :   0)
                                   :   0)  with  video_spec;

#ifndef  SIMULATE
  /*
    * Left button input bus
    */
   interface bus_in (unsigned 1) button_left ( )
     with button_white_spec;

  /*
    * Right button input bus
    */
    interface bus_in (unsigned 1) button_right ( )
     with button_black_spec;
#endif

  /*
    * Overall par
    */
  par
      {
        /*
```

```
        * VGA sync generator
        */
        SyncGen (sx,  sy,  hsync_pin,  vsync_pin);

        /*
          * Dash generated hardware
          */
        hardware ( );

        /*
          * Run-length decoder
          */
        {
        /*
          * Segment start and end positions
          */
        unsigned start, end;

        /*
          * Forever . . .
          */
        while (1)
             {
               while (sy   !=   448)
                    {
                    /*
                      * Read segment information
                      */
                   segment  ?  start;
                   segment  ?  end;

                   /*
                     * Get in the right orfer
                     */
                  if     (start > end)
                      {
                        par
                            {
                            end   =   start;
                            start   = end;
                             }
                  }

        /*
         * Make at least 1 pixel visible
         */
       if   (start  ==  end)
          end++;

        /*
          * Wait . . .
          */
        while   (sx   !=   0)
```

```
                delay;

            /*
              * Draw a scanline worth
              */
            while   (sx   !=   512)
                {
                   if   ( (sx <-   9)  >=  start  &&  (sx <- 9)  <  end)
                       {
                       video  =  1;
                       }
                   else

                       {
                       video  =  0;
                       }
                }
            }

        /*
          * Communicate button status
          */
  #ifdef  SIMULATE
                   buttons  !  1;
  #else
                   buttons  !  ~button_left.in  @   ~button_right.in;
  #endif
                  /*
                    * Wait . . .
                    */
                  while   (sy   !=   0)
                      delay;
                    }
             }
        }
   }
```

CLAIMS

1. A codesign system for producing a target system having configurable resources to provide specified functionality by:

(a) operation of dedicated hardware; and

(b) complementary execution of software on one or more software-controlled machines;

the codesign system comprising means for receiving a specification of said functionality, partitioning means for partitioning implementation of said functionality between (a) and (b) and for customising said hardware and/or said machines in accordance with the selected partitioning of the functionality.

2. A codesign system according to claim 1, wherein said partitioning means produces an optimised target system by applying a genetic algorithm to different partitions of said functionality with selection based on specified criteria.

3. A codesign system according to claim 2 wherein said partitioning means comprises means for generating a plurality of different partitions of said functionality, said codesign system comprises estimator means for estimating at least one of the speed and size of the hardware and/or software-controlled machine, and said partitioning means comprises means for selecting from said different partitions on the basis of the estimate.

4. A codesign system according to claim 1, 2 or 3 wherein the parameters of the software controlled machine are adapted during the partitioning of said functionality.

5. A codesign system according to claim 1, 2, 3 or 4 wherein said software-controlled machine comprises a processor, DSP or core.

6. A codesign system according to claim 5 further comprising means for generating a compiler for said processor, DSP or core.

7. A codesign system according to claim 5 or 6 wherein the processor, DSP or core is formed on a configurable logic circuit.

8. A codesign system according to claim 5 or 6 wherein the processor, DSP or core is formed as an ASIC.

9. A codesign system according to claim 5 or 6 wherein the processor, DSP or core is a predesigned processor.

10. A codesign system according to any one of the preceding claims, wherein the dedicated hardware is defined on a configurable logic circuit.

11. A codesign system according to claim 7 or 10 wherein the configurable logic circuit comprises an FPGA.

12. A codesign system according to any one of the preceding claims, further comprising:-

an interface cosynthesiser for defining interfaces between the hardware and software-controlled machine; and

a software compiler for compiling those parts of the functionality partitioned to software to produce corresponding machine code for execution on the software-controlled machine.

13. A codesign system according to claim 10 or any claim dependent therefrom, further comprising a hardware compiler for producing from those parts of the functionality partitioned to hardware a register transfer level description for configuring the configurable logic resources.

14. A codesign system according to claim 13 further comprising a register transfer level synthesiser for converting the register transfer level description into a netlist for configuring the configurable logic resources.

15. A codesign system according to claim 12 or any claim dependent therefrom further comprising an interface estimator for estimating the speed and area of the interfaces.

16. A codesign system according to any one of the preceding claims further comprising a width adjuster for setting a desired data word size.

17. A codesign system according to any one of the preceding claims wherein the partitioning means comprises a parser for parsing an input behavioural description of the desired functionality of the target system.

18. A codesign system according to claim17 wherein the partitioning means is adapted to respond to one of a plurality of predefined attributes in the description to perform at least one of the following:-

schedule parallel execution of processes on the hardware and on the software-controlled machine;

partition functions to software; and

partition functions to hardware.

19. A codesign system according to claim 17 or 18, further comprising means for receiving a declaration of the properties of at least one of the hardware and the software-controlled machine.

20. A codesign system according to claim 18 wherein the declaration is in an object-oriented paradigm.

21. A codesign system according to any one of claims 17 to 20 wherein the partitioning means is adapted to receive a register transfer level description of selected aspects of the target system.

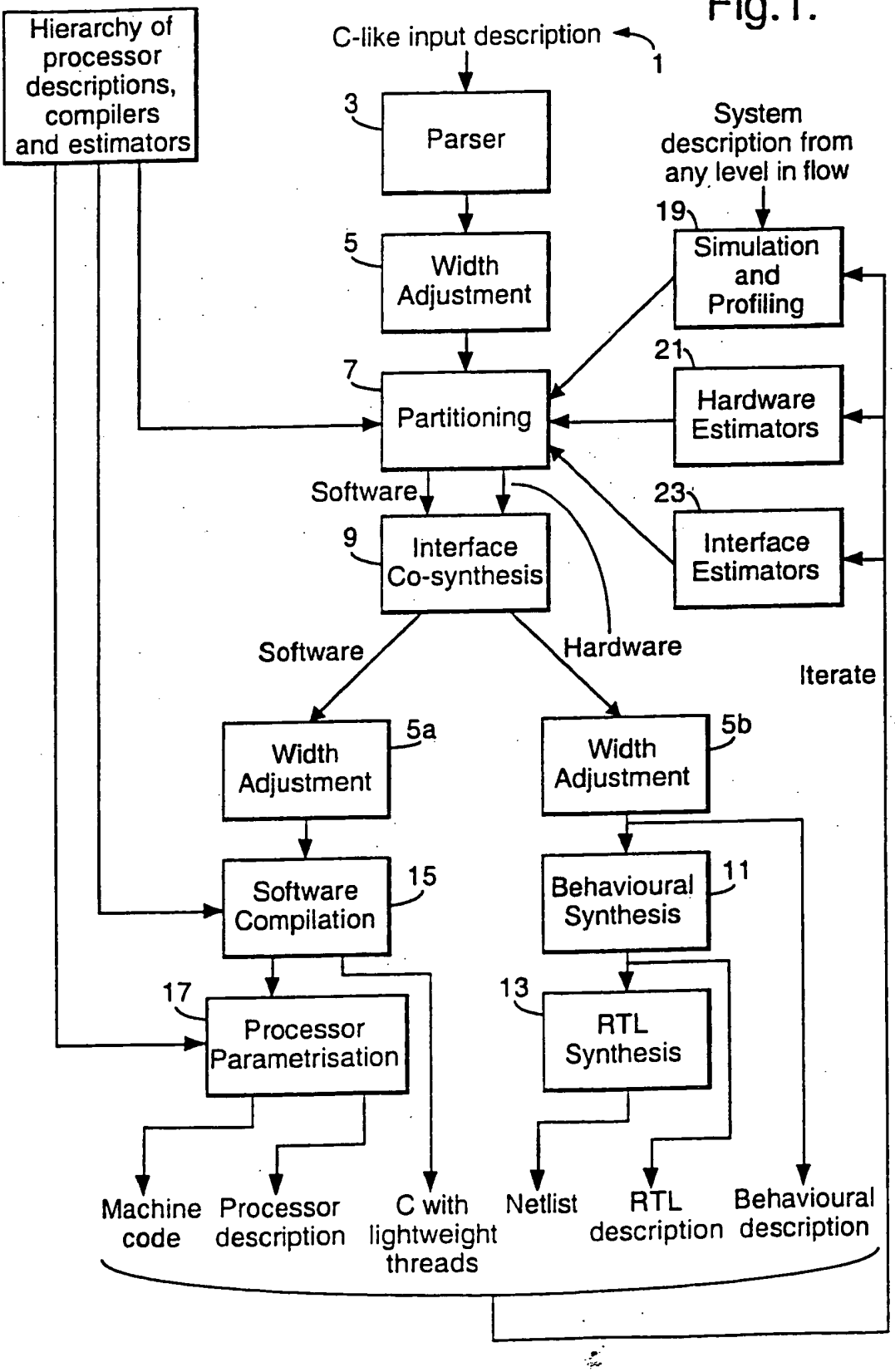
22. A codesign system according to claim 7, 10 or 11, further comprising means for configuring the configurable logic resources.

23. A codesign system according to any one of the preceding claims further comprising memory access tracing means for tracing and recording memory accesses.

24. A codesign system according to any one of the preceding claims further comprising address space partitioning means for allocating an address space to variables or functions in said specification of functionality and unifying means for unifying the address space of variables and functions with mutual operations.

25. A codesign system constructed and arranged to operate substantially as hereinbefore described with reference to and as illustrated in the accompanying drawings.

Fig.1.


Hierarchy of processor descriptions, compilers and estimators
C-like input description
1
3
Parser
System description from any level in flow
19
Simulation and Profiling
5
Width Adjustment
21
Hardware Estimators
7
Partitioning
Software
23
Interface Estimators
9
Interface Co-synthesis
Software
Hardware
Iterate
Width Adjustment
5a
Width Adjustment
5b
Software Compilation
15
Behavioural Synthesis
11
17
Processor Parametrisation
13
RTL Synthesis
Machine code
Processor description
C with lightweight threads
Netlist
RTL description
Behavioural description

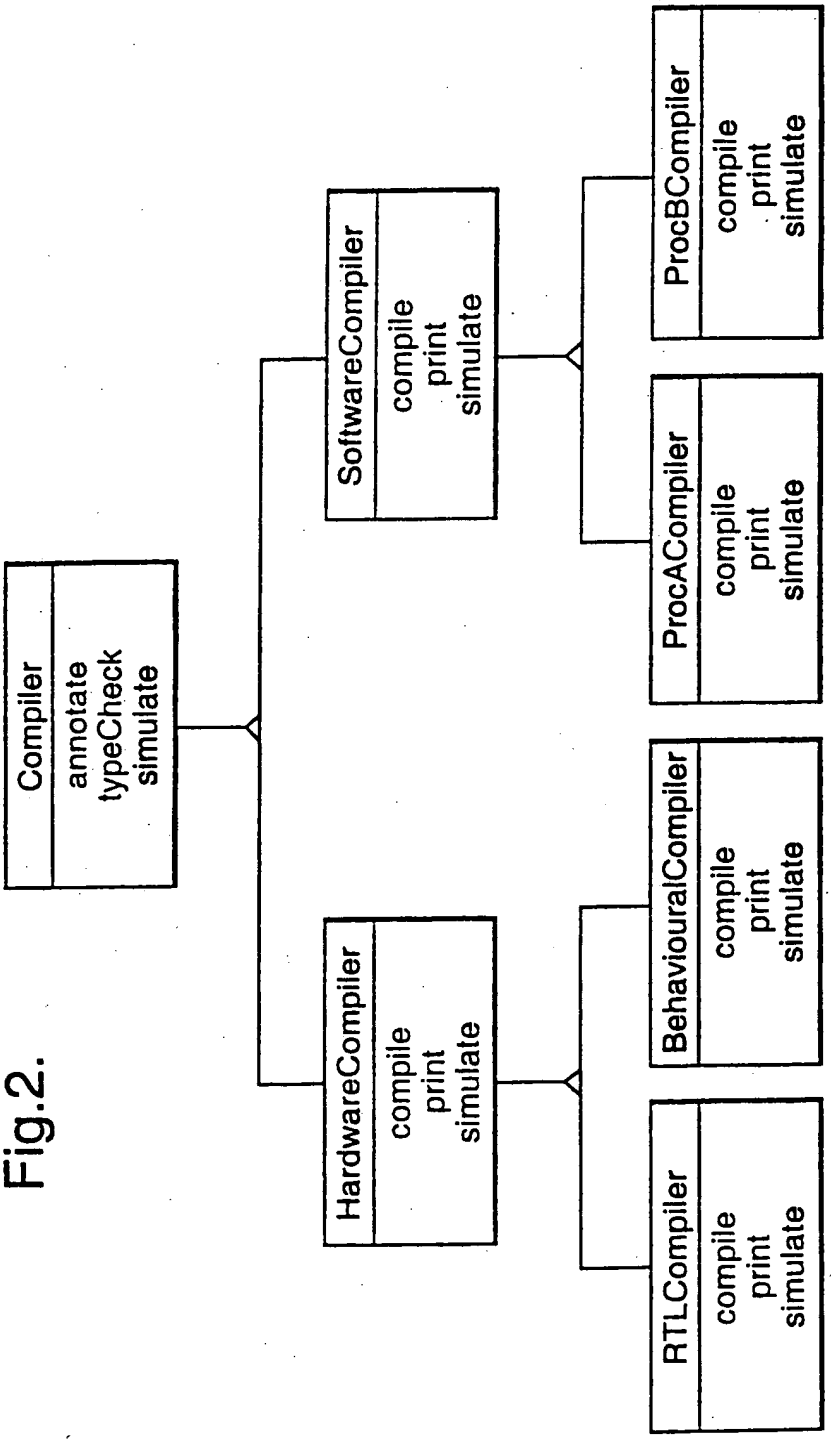

Compiler
annotate
typeCheck
simulate
SoftwareCompiler
compile
print
simulate
HardwareCompiler
compile
print
simulate
ProcBCompiler
compile
print
simulate
ProcACompiler
compile
print
simulate
BehaviouralCompiler
compile
print
simulate
RTLCompiler
compile
print
simulate

Fig.2.

Fig.3.

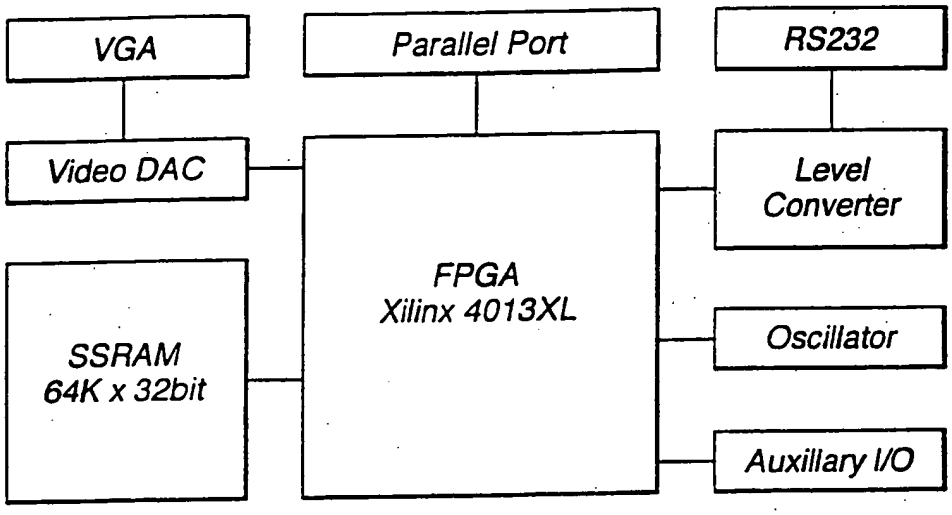

VGA
Parallel Port
RS232
Video DAC
Level Converter
FPGA
Xilinx 4013XL
SSRAM
64K x 32bit
Oscillator
Auxillary I/O

Fig.4.

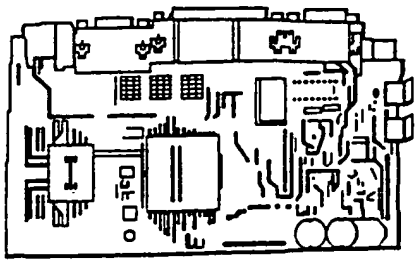


Fig.5.

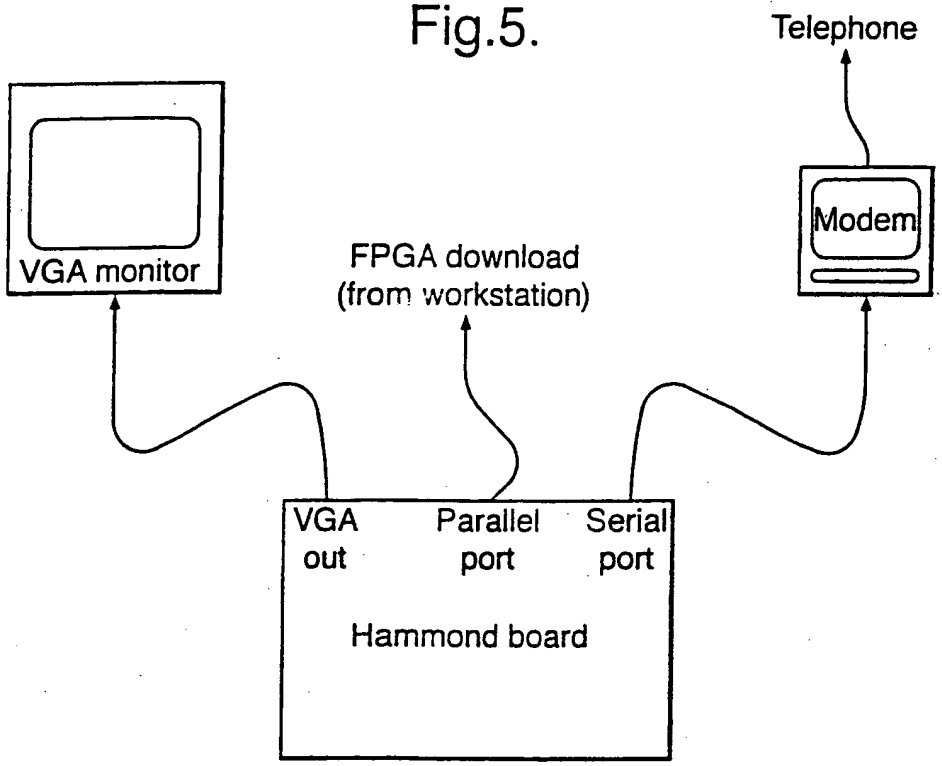
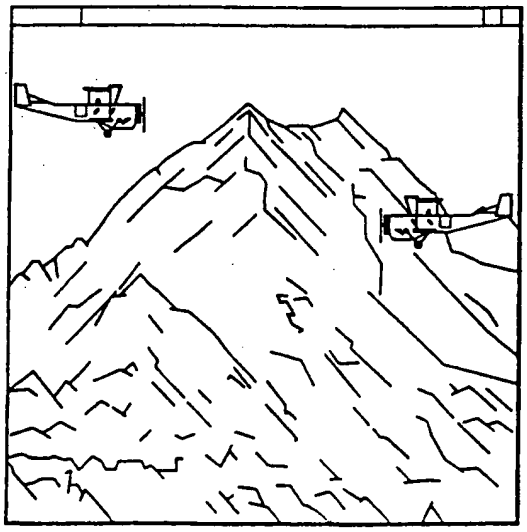

Telephone
Modem
VGA monitor
FPGA download
(from workstation)
VGA
out
Parallel
port
Serial
port
Hammond board

Fig.6.



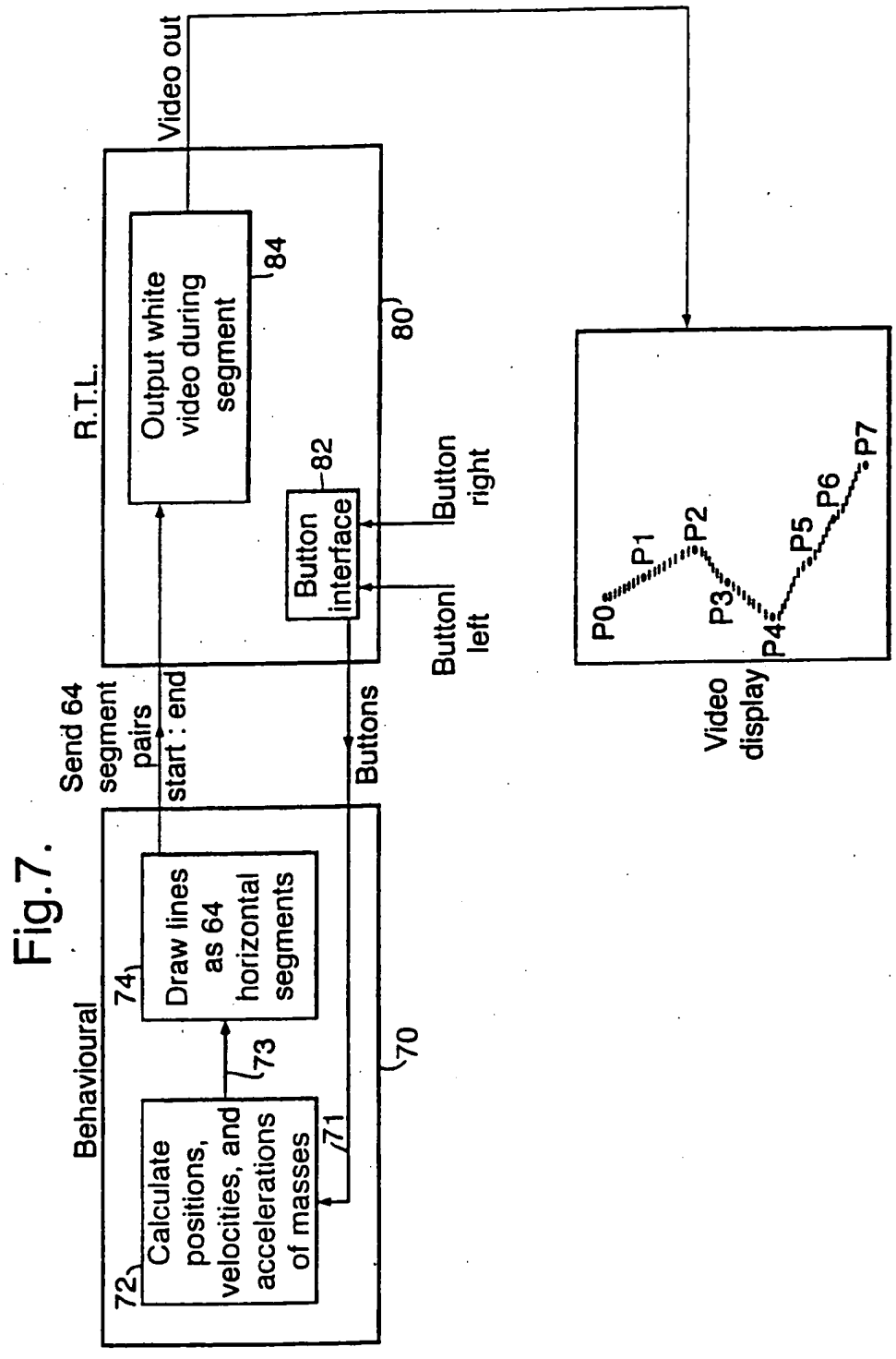

Behavioural
Send 64 segment pairs start : end
Buttons
Calculate positions, velocities, and accelerations of masses
71
72
73
Draw lines as 64 horizontal segments
74
70
R.T.L.
Video out
Output white video during segment
84
80
Button interface
82
Button right
Button left
P0
P1
P2
P3
P4
P5
P6
P7
Video display

Fig.7.

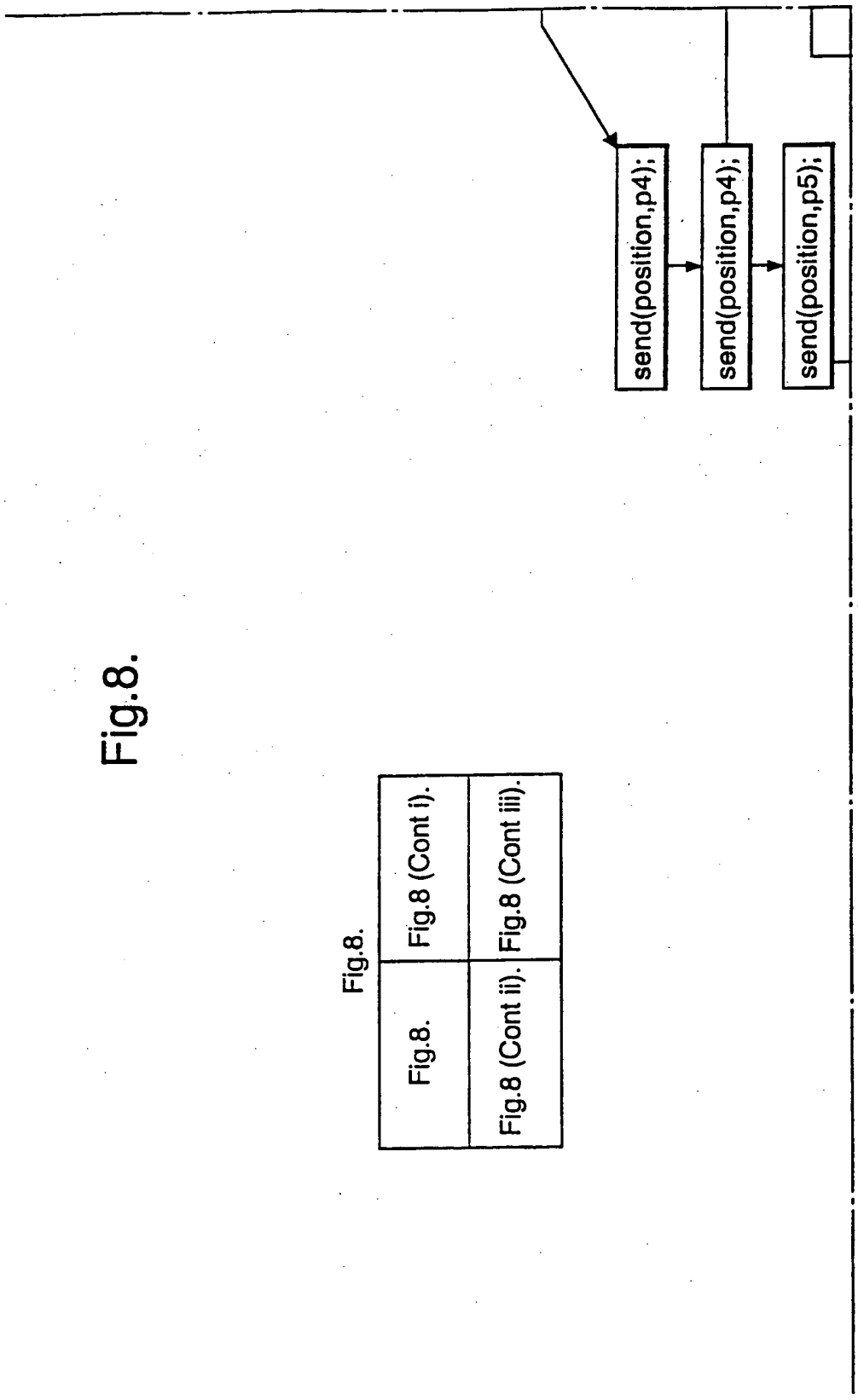
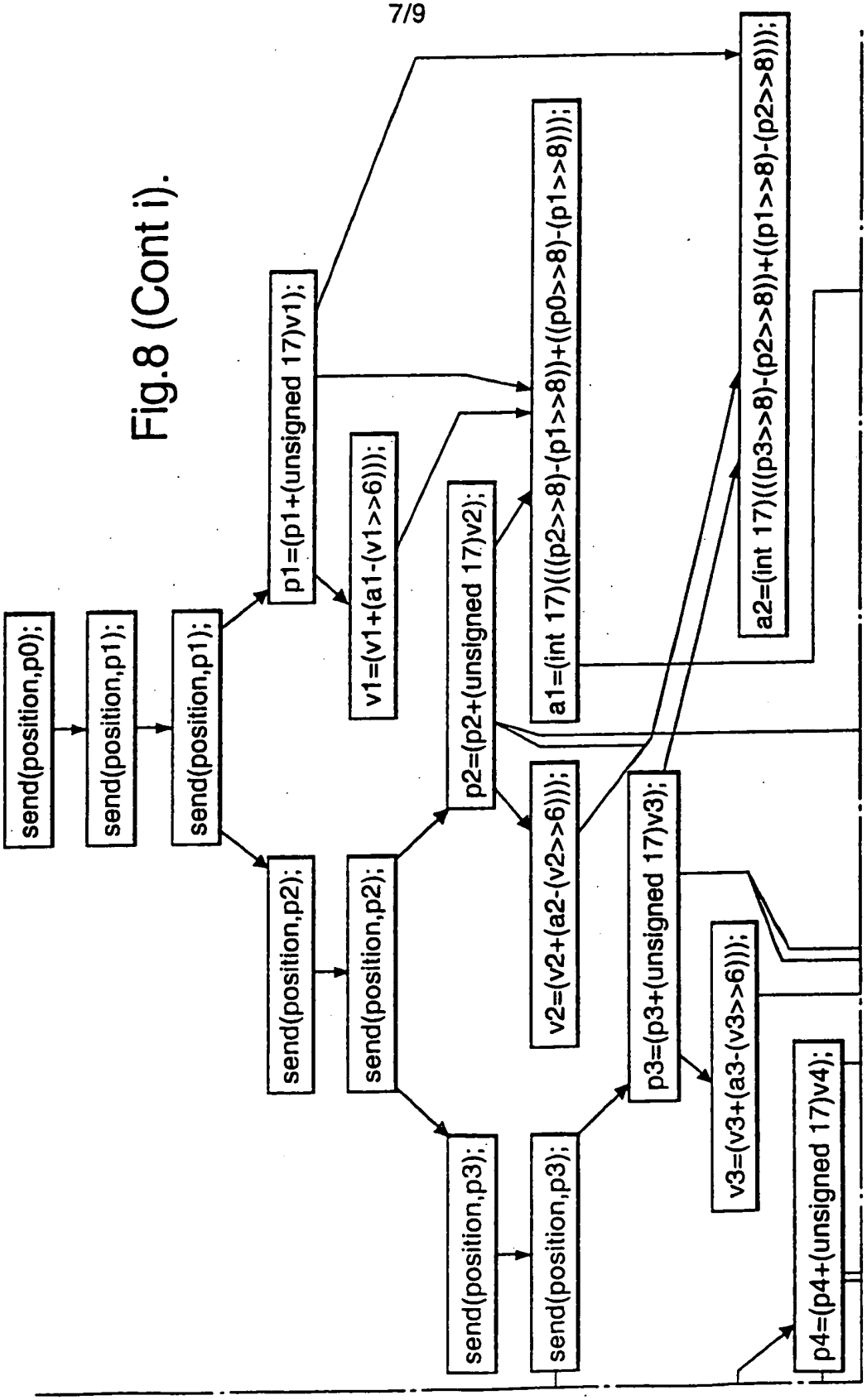


Fig.8.

Fig.8.

Fig.8.	Fig.8 (Cont i).
Fig.8 (Cont ii).	Fig.8 (Cont iii).

Fig.8 (Cont i).


send(position,p0);
send(position,p1);
send(position,p1);
p1=(p1+(unsigned 17)v1);
v1=(v1+(a1-(v1>>6)));
send(position,p2);
send(position,p2);
p2=(p2+(unsigned 17)v2);
a1=(int 17)(((p2>>8)-(p1>>8))+((p0>>8)-(p1>>8)));
v2=(v2+(a2-(v2>>6)));
a2=(int 17)(((p3>>8)-(p2>>8))+((p1>>8)-(p2>>8)));
send(position,p3);
send(position,p3);
p3=(p3+(unsigned 17)v3);
v3=(v3+(a3-(v3>>6)));
p4=(p4+(unsigned 17)v4);

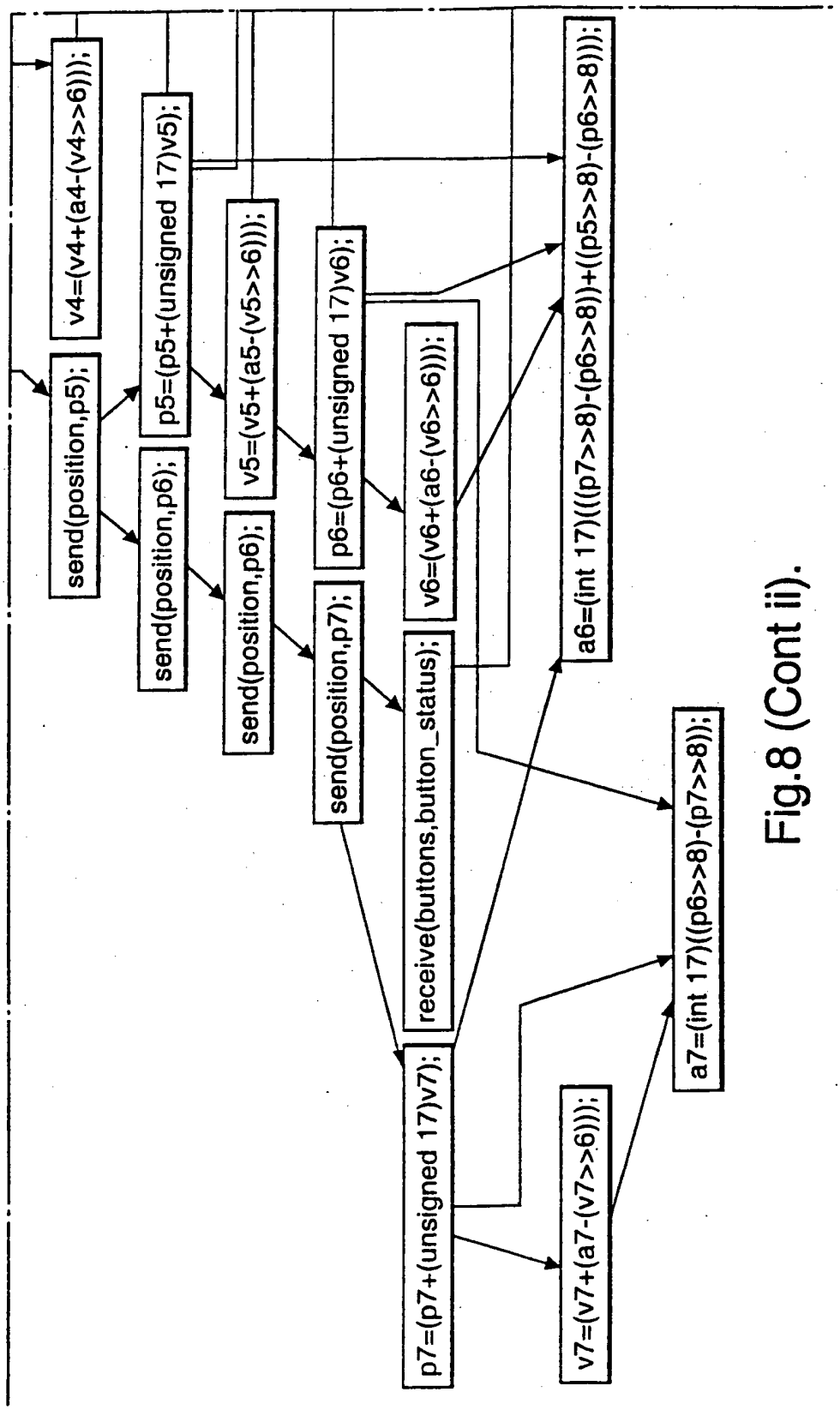

send(position,p5);
v4=(v4+(a4-(v4>>6)));
send(position,p6);
p5=(p5+(unsigned 17)v5);
send(position,p6);
v5=(v5+(a5-(v5>>6)));
send(position,p7);
p6=(p6+(unsigned 17)v6);
receive(buttons,button_status);
v6=(v6+(a6-(v6>>6)));
p7=(p7+(unsigned 17)v7);
a6=(int 17)(((p7>>8)-(p6>>8))+((p5>>8)-(p6>>8)));
v7=(v7+(a7-(v7>>6)));
a7=(int 17)((p6>>8)-(p7>>8));

Fig.8 (Cont ii).

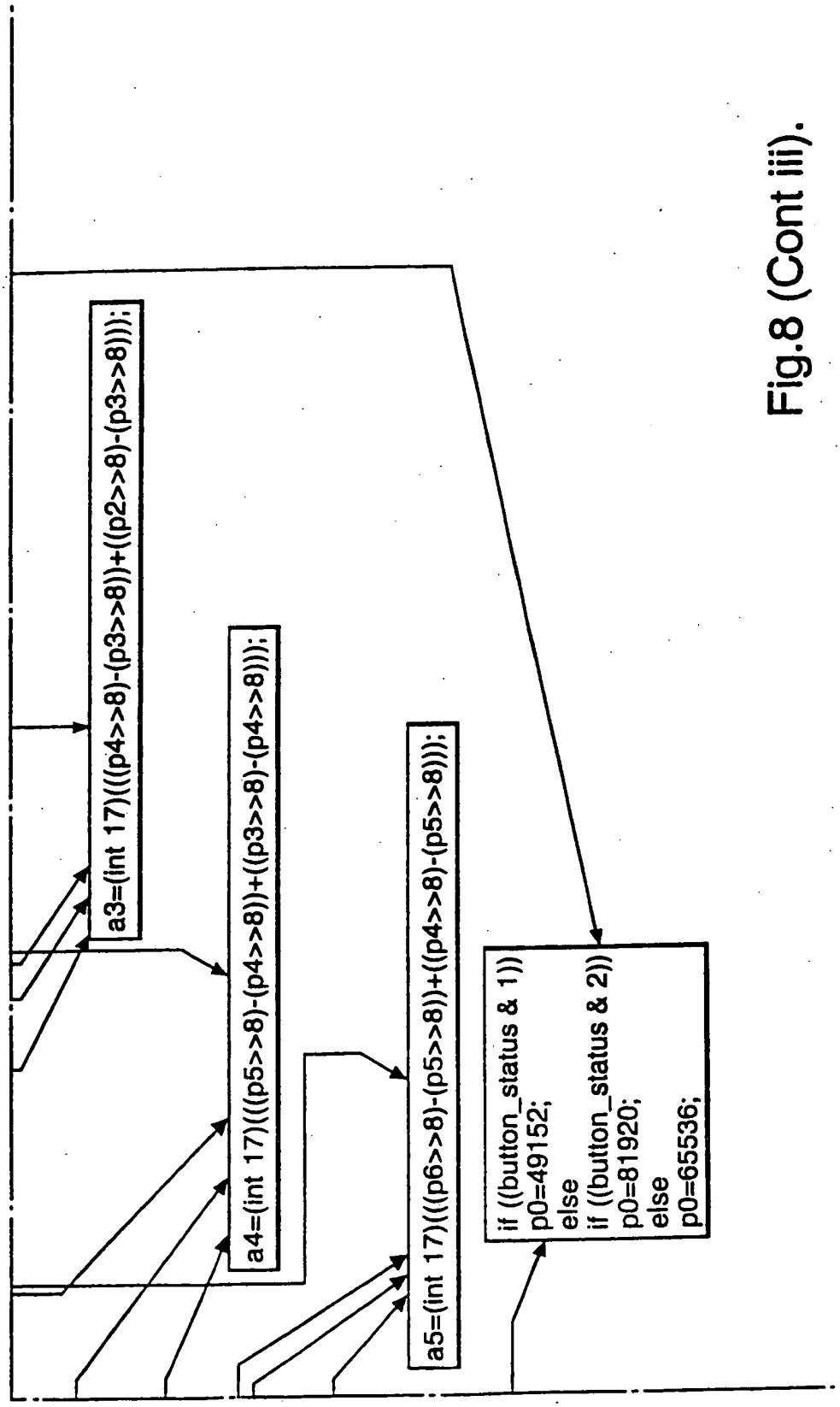


Fig.8 (Cont iii).

INTERNATIONAL SEARCH REPORT

Inter onal Application No
PCT/GB 99/04338

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F17/50

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	EDWARDS M D ET AL: "SOFTWARE ACCELERATION USING PROGRAMMABLE HARDWARE DEVICES" IEE PROCEEDINGS: COMPUTERS AND DIGITAL TECHNIQUES,GB,IEE, vol. 143, no. 1, 1 January 1996 (1996-01-01), pages 55-63, XP000554820 ISSN: 1350-2387	1,5, 9-11,13, 14,22
A	paragraphs '0001!,'0003!,'0004!; figures 1-7	3,12,17, 18,24

X	WO 94 10627 A (GIGA OPERATIONS CORP ;TAYLOR BRAD (US); DOWLING ROBERT (US)) 11 May 1994 (1994-05-11)	1,5, 9-11,13, 14,22
A	page 8, line 5 - line 23 page 35, line 5 -page 41, line 11 figures 17,21	17,18,24

	-/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
24 May 2000	09/06/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31–70) 340–2040, Tx. 31 651 epo nl, Fax: (+31–70) 340–3016	Guingale, A

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Inter ional Application No
PCT/GB 99/04338

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	WO 97 13209 A (ERICSSON TELEFON AB L M ;ANDERSSON ROEJAAS KARL GUNNAR (SE)) 10 April 1997 (1997-04-10) abstract page 2, line 8 -page 4, line 3	6,8,17, 19
A	EP 0 772 140 A (IMEC INTER UNI MICRO ELECTR) 7 May 1997 (1997-05-07) abstract column 13, line 6 - line 55 column 19, line 24 -column 24, line 41 figures 2,6,7	1,9, 12-14, 17,18,24
A	LECHNER E ET AL: "The Java Environment for Reconfigurable Computing" INTERNATIONAL WORKSHOP ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, FPGAS,GB,ABINGDON, 1 September 1997 (1997-09-01), pages 284-293, XP002086682 paragraph '0002! figures 1,2	1,7,10, 11
P,X	SAUL J M: "Hardware/software codesign for FPGA-based systems" PROCEEDINGS OF THE 32ND ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEMS SCIENCES. 1999. HICSS-32. ABSTRACTS AND CD-ROM OF FULL PAPERS, PROCEEDINGS OF HICSS 32 - 32ND ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, MAUI, HI, USA, 5-8 JAN., page 10 pp. XP002138152 1999, Los Alamitos, CA, USA, IEEE Comput. Soc, USA ISBN: 0-7695-0001-3 the whole document	1-25

Form PCT/ISA/210 (continuation of second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Information on patent family members

Inter ional Application No
PCT/GB 99/04338

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 9410627 A	11-05-1994	US 5535342 A	09-07-1996
		AU 5458194 A	24-05-1994
		CA 2148813 A	11-05-1994
		EP 0746812 A	11-12-1996
		JP 8504285 T	07-05-1996
		AU 5593594 A	24-05-1994
		CA 2148814 A	11-05-1994
		EP 0667010 A	16-08-1995
		JP 8504514 T	14-05-1996
		WO 9410624 A	11-05-1994
		US 5497498 A	05-03-1996
		US 5603043 A	11-02-1997
		US 5857109 A	05-01-1999
WO 9713209 A	10-04-1997	SE 505783 C	06-10-1997
		AU 7233196 A	28-04-1997
		CA 2233843 A	10-04-1997
		CN 1202972 A	23-12-1998
		EP 0853792 A	22-07-1998
		JP 11513512 T	16-11-1999
EP 0772140 A	07-05-1997	EP 0767544 A	09-04-1997
		US 5870588 A	09-02-1999